Mid-term. May 4, 2015

CS180: Algorithms and Complexity
Spring 2015

Guidelines:

- The exam is closed book and closed notes. Do not open the exam until instructed to do so. You will have to stop writing at 5:50 PM.
- Write your solutions clearly and when asked to do so, provide complete proofs. You may use results we proved in class without proofs as long as you state what you are using. You may also use any algorithms we covered in class as sub-routines without giving full descriptions (or rewriting the whole algorithm again).
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly.

Problem	Points	Maximum
1	3.5	5
2	10	10
3	10	15
4	0	25
5	25	25
6	10	10
7	23	30
Total		120

Name:
UID:
Section:

3,5

1 Problem

For each pair (f, g) below indicate the relation between them in terms of O, Ω, Θ. For each missing entry, write-down Y (for YES) or N (for NO) to indicate whether the relation holds (no need to justify your answers here). For example, if $f = O(g)$ but not $\Omega(g)$, then you should enter Y in the first box and N in the other two boxes. Similarly, if $f = \Theta(g)$, then you should enter Y in all the boxes. [5 points].

$f = O(g) = \Omega(g) = \Theta(g)$

f	g	O	Ω	Θ	
$\log_2 n$	$\log_{10} n$	Y	Y	Y	
$2^{(\log_2 n)^4}$	n^5	Y	N	N	-0,5
$n^3 \cdot 2^n$	3^n	Y	N	N	
$2\sqrt{\log_2 \log_2 n}$	$\log_2 n$	N	Y	N	-0,5
$n!$	n^n	N	Y	N	-0,5

2 Problem

10

Answer true or false for the following (no need for explanations) [10 points]:

- Consider an instance of the stable matching where a doctor D_1's first choice is H_1 and H_1's first choice is D_1. Is it true that in every stable matching D_1 should be matched to H_1?

  $D_1 \; (H_1, H_2) \; (D_1, D_2) \; H_1$

  $D_2 \; (H_1, H_2) \; (D_2, D_1) \; H_2$

  True

- Consider an instance of the stable matching problem and a candidate perfect matching M where one doctor gets her top choice and one hospital gets its top choice, while every other doctor and hospital get their second choice. Is M necessarily a stable matching?

  False

- If α is an n'th root of unity, then $\sum_{j=0}^{2n-1} \alpha^j = 2n$.

  $2n-1-7$

  $n=4$

  $\alpha = 1, i, -1, -i$

  $i^0 + i + i^2 + i^3 + i^4 + i^5 + i^6 + i^7 = 8?$

  $1 + i - 1 - i + 1 + i - 1 - i = 0$

  $1, \omega, \omega^2 \cdots \omega^{n-1}$

  $\hookrightarrow 1^0 + 1^1 + 1^2 + \dots + 1^{2n-1} = 2n$

  if $n=1$

  $n \omega \; \omega^0 + \omega^1 + \dots + \omega^{2n-1} = 2n$

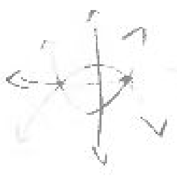
  False

- Dijkstra's algorithm when run on an unweighted undirected graph starting from a vertex s gives us (by looking at the graph formed by *Parent* links) the breadth-first-search tree starting from s.

  True

- Any polynomial $P : \mathbb{R} \to \mathbb{R}$ of degree d is uniquely determined by its evaluations at d distinct points $x_1, \dots, x_d$.

  $d+1$



  False

10

A: B
B: A C D
D: B E
C: B
E: D

3 Problem

Given a connected undirected graph $G = (V, E)$ as input (in adjacency list representation), give an algorithm to check if G is a tree. You must analyze the time-complexity of your algorithm but don't need to prove correctness. For full credit, your algorithm should be correct and run in time $O(|V| + |E|)$. [15 points]

Algorithm:

- Using the BFS algorithm, generate BFS tree called "T"
- If the set of edges in G (E) = the set of edges in T (E'), return true
  else return false

how do you check this?

BFS runs in $O(|V|+|E|)$. We must check if every edge in G remains in T after BFS, which is $O(|E|)$ if done efficiently with the adjacency list representation (when checking $e \in E$, use its vertices as indices in E', which should be O(1) array access).

4 Problem

Given the coefficients of a polynomial P of degree d and an integer k as input, give an algorithm to compute the coefficients of the polynomial $P(x)^k$. For example, if your input is $(1,1)$ (to denote the polynomial $1+x$) and $k=3$, your output should be $(1,3,3,1)$ to denote the polynomial $(1+x)^3 = 1+3x+3x^2+x^3$. Similarly, if the input is $(1,-3)$ (to denote $P = 1-3x$), $k=3$, your output should be $(1,-9,27,-27)$.

To get full credit, your algorithm should be correct, run in time $O((k \cdot d)\log(k \cdot d))$ and you must analyze the time-complexity of your algorithm (no need to prove correctness). [25 points]

Remark: Here we measure time-complexity as in the fast-polynomial multiplication algorithm, where we count complex additions and multiplications as unit-cost.

X

$\boxed{0}$

5 Problem

Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is an *ordered graph* if it has the following properties.

Rule 1. Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

Rule 2. Each node except v_n has at least one edge leaving it. That is, for every node v_i, $i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

Given an ordered graph $G = (V, E)$ (in adjacency list representation), give an algorithm to compute the number of paths that begin at v_1 and end at v_n.

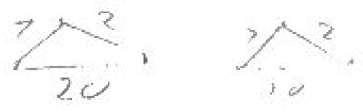
You must analyze the time-complexity of your algorithm (no need to prove correctness). To get full-credit your algorithm must be correct and run in time $O(|V| + |E|)$. (25 points)

Remark: You can assume that adding two numbers takes constant time in your time-complexity calculations.

```
· Start with an array Results[|V|] initialized to -1
· Call NumPaths(1,n) for solution
· NumPaths(i, n):
    · If Results[i] ≠ -1, return Results[i]   // Memoization
    · output = 0;
    · If i == n, output = 1;
    · Else for each node v_j adjacent (directed) to v_i:
        · output = output + NumPaths(j, n)   // O(1)
    · Results[i] = output   // memoization
    · return output
```

25

This algorithm uses memoization to prevent recomputation. NumPaths in the worst case (straight line 1->n) must be calculated for every vertex and edge, since it will follow the directed edges of vertices it encounters to encounter more vertices. But a vertex can only be visited once because of Rule 1 (no backtracking) and computed once because of memoization. We thus never use an edge more than once. Since array operations and addition are O(1), NumPaths(1,n) is O(|V| + |E|).



10

6 Problem

Decide whether the following statement is true or false. If it is true, give a short explanation (no need for a formal proof - a high-level description is enough). If it is false, give a counter-example.

Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? T must still be a minimum spanning tree for this new instance. [10 points]

True ✓

Consider Kruskal's ✓ algorithm for minimum spanning trees, which is proven to produce MST, including T of G. Let G' be the new graph where $c_{e'} = c_e^2$.

Kruskal adds the least safe edge e every iteration according to its cost. Consider arbitrary edges a and b, where in G $c_a < c_b$ ✓ (all edge costs >0 and distinct). Then in G', $c_a^2 < c_b^2$ holds ($x^2 < y^2$ if $x<y$ and $x, y > 0$). So Kruskal will add a before b to T, whether its input is G or G'. Thus Kruskal will again produce T for G', meaning T is still a valid MST of G'.

7 Problem

Given an undirected graph $G = (V, E)$, a subset of vertices $I \subseteq V$ is an independent set in G if no two vertices in I are adjacent to each other. Let $\alpha(G) = \max\{|I| : I \text{ an independent set in } G\}$. The goal of the following questions is to give an efficient algorithm for computing an independent set of maximum size in a tree. Recall that a *leaf* in a graph is a vertex of degree at most 1 and that every acyclic graph (graph without any cycles) has at least one leaf.

Let $T = (V, E)$ be an acyclic graph on n vertices. a tree

1. Prove that if u is a leaf in T, then there is a maximum-size independent set in T which contains u. That is, for every leaf u, there is an independent set I such that $u \in I$ and $|I| = \alpha(T)$. [15 points]

2. Give the graph T as input (in adjacency edge representation), give an algorithm to compute an independent-set of maximum size, $\alpha(T)$, in T. To get full credit your algorithm should run in time $O(|V| + |E|)$ (or better) and you ~~must prove~~ correctness of your algorithm. You don't need to analyze the time-complexity of your algorithm and it is sufficient to solve this problem assuming part (1) (if you want) even if you don't solve it. [15 points]

not all

① Assume: I is a maximum-size independent set in T that does not contain u. Then there are two possibilities:

(a) A vertex j in I is adjacent to u. Then j is the only vertex adjacent to u, since u is a leaf. We can substitute j for u to create I_1 of the same size as I. u cannot be adjacent to other members in I or I_1, since it was only adjacent to j, so I_1 is still an independent set. Thus $u \in I_1$, a maximum-size independent set ✓

15

(b) No vertex in I is adjacent to u. Then u can be added to I_1, increasing its size by 1 while retaining independent set status. This contradicts our assumption that I is maximum-size independent set, so this "possibility" is actually impossible.

Thus: if I is a maximum-size independent set that does not contain a leaf u, it is always possible to construct I_1 of equal size containing u. All graphs have at least one I, so there exists I_1 such that $u \in I_1$ and $|I_1| = \alpha(T)$. □